Night of the Lepus
A Plan 9 Perspective on Blue Gene's Interconnects

Charles Forsyth [a]
Jim McKie [b]
Ron Minnich [c]
Eric Van Hensbergen [d]

ABSTRACT

The nodes of a Blue Gene supercomputer are connected by five networks, four of unusual design. Only the I/O nodes have a conventional IP-based Ethernet. Plan 9's internal and external representations of networks is simple and clean yet powerful enough to encompass those networks directly, without having to emulate existing network types. In supercomputer environments, applications often use the network hardware directly, but that makes it hard or impossible to share. We wish system services to make good use of the specialized networks too, in order to build fault-tolerant distributed services. Our kernel must therefore mediate access to those networks, just as it does for IP traffic.

Introduction

The Blue Gene supercomputer[1] has five different concurrent interconnects: a conventional Gigabit Ethernet, a class routed network interface capable of collective operations, a 3D torus, a low-latency barrier and signal network, and a management network primarily used for initial system load and monitoring. The Blue Gene/L (BG/L) model is comprised of up to 65,536 compute nodes and 1,024 I/O nodes partitioned in up to 1,024 logical processing sets (or *psets*). Each pset has an I/O node servicing up to 64 compute nodes. Compute nodes are connected together by the class routed network. I/O nodes are connected to the outside world by the Gigabit Ethernet, and are connected to the class routed network but not the torus. All nodes inside the Blue Gene are connected to the management and barrier networks. (See Figure 1.)

An external service node is connected to the I/O nodes via the Gigabit Network and is also connected to the management network through a specialized gateway. In practice, I/O nodes typically run Linux and compute nodes run the specialized Compute Node Kernel (CNK). Applications on compute nodes either directly access the interconnect resources or do so via an HPC communications framework such as MPI.[2]

As part of a project[3,4] exploring alternatives in distributed systems infrastructure on ultrascale platforms, we have ported Plan 9 from Bell Labs[5] to the Blue Gene/L platform. Our goal is to evaluate and extend Plan 9's principles to work in tightly coupled environments with thousands to millions of cores, with the intent of broadening the application base for such machines while making them more approachable to developers and end-users. We believe that Plan 9 is a 'right weight kernel'[4] that balances the trade-offs between spartan light-weight-kernels and plump elaborate operating systems such as

[a] – Vitanuova Ltd — [b] – Bell Laboratories — [c] – Sandia National Labs — [d] – IBM Research

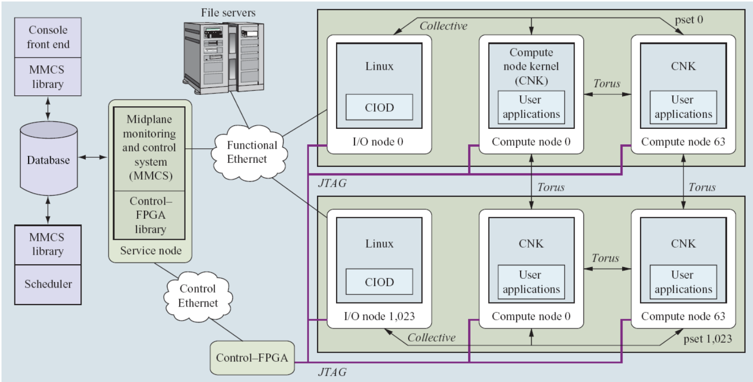


Figure 1. Structure of Blue Gene and a few nodes.

Linux. Furthermore, we can experiment with new models of system and application organization that exploit Plan 9's unique perspective on distributed resource sharing and management.

To do that, we first needed working networks on BG/L. After porting and stabilizing the Plan 9 kernel on Blue Gene hardware, our next step was to develop device drivers and interfaces for the various interconnects. This paper describes this initial design and discusses possibilities for future work.

Design Overview

Plan 9 represents system resources, including devices and networks, by files and directories in dynamically-created *name spaces*.[6,7] Those name spaces are implemented by *file servers*, which can be built-in to the kernel like a conventional device driver, or implemented as an otherwise ordinary user-level application serving a lean file service protocol, 9P. The names and contents of a name space — the files and directories in it — can be synthesized on the fly; they need not exist on any permanent storage (unlike, say, the 'special files' of Unix).

Simple devices in Plan 9 are typically represented by a pair of files: one carrying data in any convenient text or binary format, and one conveying in text form commands to control the device. More complex devices use directory hierarchies. For instance, by convention, a multiplexing device has numbered subdirectories for each *conversation* (eg, an interface, a protocol, or a remote address). The device has a `clone` file that, when opened, atomically creates a new conversation directory. The specific port (protocol, etc) is then selected by writing the text "`connect` *address*" into a control file in the resulting directory. There are similar conventions for listening for incoming calls. Keeping strictly to a few documented conventions, including representing all network addresses in text not binary form, allows applications to interact with any type of network through a small set of library functions — `dial`, `announce`, `listen`, `accept` and `reject` — without having to know network-specific details (including its addressing syntax or structure). Even those library functions simply manipulate files and directories in a formal way.

Thus, in Plan 9 we naturally represent all five interconnects listed above by various forms of name space, and indeed, they suggest something of the range of possibilities that representation allows. Furthermore, because devices, networks and their interfaces are all represented in the name space, they can be implemented either in the kernel or by a 9P server. For example, the name space for barriers is provided by a Plan 9 device driver, compiled into the CPU and I/O kernels. By contrast, the name space for low-level node management is implemented by a user-level application that does not run on

the Blue Gene nodes itself, but on an external Plan 9 system. It securely exports that name space for use anywhere on the network.

The other interconnects are all general-purpose messaging networks, and thus closer to each other in style and implementation, but they still illustrate some important differences, because of the underlying differences in network structure. We use them as examples of how straightforward it is, in Plan 9, to use a new network type as an Internet interface, without preventing the use of a network's specialized abilities by non-Internet applications. The name spaces representing these networks are used by applications directly, but they are also used by other drivers that implement higher-level devices or protocols.

Barriers and signals

From an application's point of view, a *barrier* controls transitions between phases in a distributed computation: a node arriving at a barrier cannot proceed until all other nodes have reached the barrier. The global *signal* by contrast is true if any node has set the signal, and might be used by one node to announce 'Eureka!', or to notify all other nodes of failure.

The global barrier and signalling hardware provides four identical networks, each combining a single bit from every node, wired into a tree. Each network can be independently programmed to form the global AND or the global OR of those bits. All nodes must agree on the Boolean operation. The calculation includes only bits from nodes that have enabled the interface, and subtrees of nodes can be enabled or disabled under control of external management software, to isolate psets and faulty nodes. Each node can set its interface to give an interrupt depending on the value of the resulting bit.

In Plan 9, on each node the device driver represents the interface for each network *n* ($0 \leq n \leq 3$) by two files synthesized by the device driver:

`gib`*n*`barrier`	global AND
`gib`*n*`signal`	global OR

The files are exclusive-use, and only one can be open at a time, since the barrier and interrupt functions of the network are mutually exclusive. When either file is opened for a given network, the hardware is prepared for the appropriate Boolean operation.

The actual operations are triggered by read and write system calls. Barrier operation is simple: a write (with arbitrary data) blocks the writing process until all other nodes with the barrier active have also arrived at the barrier, and then the write returns. For example, a program might use the barrier file as shown in Figure 2. When an operation is performed, the kernel can optimistically opt to poll for a few microseconds, and only then wait on the more expensive interrupt.

The `gib`*n*`signal` file is similar. It is written (again with arbitrary data) to post an signal on the network. A process waits for a signal simply by reading the file. Each read on the file blocks until a signal is detected; a signal has occurred when the read returns.

The example above was in C but, unusually for an HPC programming interface, the file-based interface allows operations to be invoked directly in a shell script by familiar existing commands such as `echo` or `cat`. For example at boot time, the CPU nodes might need to wait for some global event before proceeding further in the system initialization script `/bin/cpurc` or they might need to signal a catastrophic initialization failure before any other network is ready to do so. Use in a shell script is straightforward, but the script must open the file well before use; it might be better to enable the interface and mode through a control file, allowing a simple `echo >/dev/gib0barrier` to mark each barrier.

```
int barrierfd;

void
openbarrier(void)
{
	barrierfd = open("/dev/gib0barrier", ORDWR);
}

void
barrier(void)
{
	/* mark our arrival at the barrier */
	write(barrierfd, "", 1);
	/* on return, all others have reached and may now pass the barrier */
}

void
compute(void)
{
	openbarrier();
	phase1();
	barrier();
	phase2();
	barrier()
	...
}
```

Figure 2. Using a barrier file.

For an application to use this device, all processes running on all nodes must know which of the four interfaces to use, and all must use it in the same mode. Currently, the selection is done out of band, and the application is given the instance when it starts. If the need arises, we might provide a 9P service for allocation and release of barriers. The same service could also allocate the global IDs needed for other networks such as the Torus and Tree.

The kernel has its own internal interface to the barrier, because initialization of the Torus and Tree networks must be synchronized with other nodes, as discussed below.

Monitoring network

Owing to the scale of the system, Blue Gene incorporates a special purpose network to facilitate boot-strap, monitoring, and debug of the entire system. The monitoring network comprises a number of management chips that connect to the front-end management system over a 100 Mbit Ethernet link and talk to the Blue Gene nodes over their JTAG interfaces. The primary mechanism used to transfer data is the 16k SRAM available on every node along with some signalling bits available in device control registers. System control software uses the SRAM to do the initial software load into the Blue Gene, then broadcast along the Ethernet network to simultaneously load all nodes of the system. When software load is complete, the management system continues to monitor status through JTAG access of the SRAM. Since nodes have no externally accessible serial ports, consoles are redirected through buffer mailboxes available in the SRAM.

Our Plan 9 infrastructure represents the external interface to this JTAG infrastructure as a file system. This file system is served by a special application on an external system that connects to a management port on the service node which accepts JTAG and system management commands. Directory hierarchies are used to organize racks, midplanes, node cards, nodes, and cores. Each core has its own subdirectory with a set of files representing the core's state and control interfaces.

The node's buffer mailboxes are represented as two synthetic files that look and act like standard Plan 9 serial ports — as such, developers can access the console as if they were directly connected over a serial line. The buffer mailboxes can be configured to be blocking so that system activity blocks until they are read (which is useful during bring-

up when missed messages may be critical). A special log file is also provided which uses a rotating character buffer to store the last 1k of console activity in the likely case that an interactive console isn't desired (indeed, with 64k nodes, that would be a lot of interactive consoles). This log buffer is useful for assessing the cause of panics and other system error conditions.

Many of the other files in the per-core directory look like entries in the Plan 9 `/proc` file system. These give access to the system's current registers, memory, status, and loaded executable image. Since these files' syntax and semantics are identical to normal application process formats, they can be used by the Plan 9 debuggers (`acid`[8] and `db`)[9] to debug running kernels. Even better, `acid` was written to be a multi-threaded parallel debugger, and can therefore debug several cores in parallel — obviously very useful on such a large machine. Furthermore, `acid` includes a programmable debugging language, allowing new cross-node operations to be defined. Messages to the control file can be used to stop, start, and single-step the cores. The control file also provides mechanisms for setting the nodes boot image, as well as rebooting all allocated nodes. Supplemental files are used to expose access to special purpose registers not normally available in the context of user processes.

Within each node, the kernel makes the monitoring network visible as two serial ports via Plan 9's normal UART driver. They can be used as system consoles, or for any other application one might wish to use a serial port. The SRAM memory itself may also be accessed through reads and writes to a synthetic file in `/dev`. Aspects of the node's configuration in SRAM are extracted by a driver and presented as separate text files, such as the machine's designated IP address, torus coordinates, and 128-bit security key, making them easy to access from both C programs and shell scripts. (An example of their use appears later.)

The monitoring network and debug file system were vital contributors to the speed with which we were able to bring up the system. Additional extensions currently being planned include making available the pre-configured 'personalities' of each node, as well as visualizations of the network routing topologies. New subdirectories are planned to provide more readable access to device status and control for debug purposes. And hooks are planned for configuring and accessing performance counter data while the system is running.

A further extension is planned to support more efficient debugging of large system runs. Although `acid` is able to cope with multiple threads of execution, we wish to coordinate debugging across the system, and across different aspects of the system. Managing 64k or 128k processes directly through `ps` or `acid` is clumsy. We shall therefore provide an aggregation service to allow core or node groups to be established to allow coordinated halt, start, and single-step of large portions of the system. This grouping mechanism will take the form of a clone-based synthetic file hierarchy, with numbered subdirectories representing current group of cores, and a clone file to create new subdirectories to configure new groups, in just the same way as new interfaces and connections are created in `/net`. Commands issued to the control files in those subdirectories will be issued simultaneously to all their members. Finally, we plan on extending the monitoring facilities to allow more complete reports of system status through SRAM locations, as well as exposing publishing interfaces to middleware, and applications are planned to allow for more complete external monitoring of the system as a whole, similar in principle to Supermon.[10]

Ethernet

Each I/O nodes on Blue Gene has a standard Ethernet interface — a variant of the PowerPC 440's EMAC — that links it to physical storage, external networks, and ultimately to front-end nodes that launch user applications. (There is no Ethernet on CPU nodes.) The software interface is the same as for Ethernet on any other Plan 9 system, but is

worth describing briefly as an example of name spaces for multiplexors.

Ethernet devices are conventionally bound under the /net directory in the name space,[11] with each available Ethernet interface in its own directory (ether0, ether1, etc.). Each interface directory contains files revealing the MAC address, Ethernet statistics, and interface-specific statistics. It multiplexes the packets of different Ethernet protocols sent and received on that interface: a numbered subdirectory represents a particular Ethernet protocol (ie. IP, ARP, etc.). Following the usual Plan 9 conventions for multiplexors, opening the clone file makes a new protocol directory, and writing a connect request to its control file selects the protocol type. Packets of that type are read and written on the data file.

The existing Plan 9 Ethernet driver has two levels. The top portion, essentially machine-independent, presents the name space for each Ethernet interface, and does the multiplexing.[11] It connects each interface directory to a low-level driver that is the source and sink for packets on that interface.

We modified Plan 9's existing low-level EMAC driver to work with the BG/L's variant. To extract maximum performance, we use the gigabit mode, which includes jumbo frame support. Unfortunately the hardware limits the size of buffers to 4k bytes. This somewhat complicates the device driver which must chain several DMA buffers together in order to obtain jumbo-frames. Obtaining optimal performance will be even more difficult on the BG/P platform, which uses 10 Gigabit Ethernet interfaces on the I/O nodes.

Torus

The three-dimensional torus network is the main interconnect for CPU nodes. The nodes are addressed on the torus by their *x*, *y*, and *z* coordinates, available in the node *personality* at boot. The personality also contains information about the dimensions of the torus (or mesh if it is not wrapped at the edges) which is used by the O/S driver to initialize the hardware. The network is reliable, flow-controlled, and does adaptive routing, all in hardware. Packets can, however, arrive out of order. Each node is connected to its six nearest neighbors with independent bi-directional bit-serial links clocked at 1.4Gb/s; the effective bandwidth of an individual link is 155MB/s after hardware link overhead. There is approximately 100ns latency to cut-through an intermediate node as a packet is routed to its destination.

Injecting packets into the network and receiving packets from the network is done via 128-bit wide memory-mapped FIFOs. Each *group* of FIFOs is replicated; there is no difference between the two sets other than their naming. There are seven receive FIFOs in each group, one for each direction (*x+*, *x–*, *y+*, *y–*, *z+*, *z–*) and one priority (priority is indicated by a bit in the packet header and is used by the hardware routing). An injection FIFO group consists of three general FIFOs and one priority FIFO; packets may be injected into any of the FIFOs regardless of the direction of their destination.

The granularity of the FIFOs and of packets in the network is 32-byte *chunks*. Packets are variable length, one to eight chunks. There is an eight-byte packet header at the beginning of the first chunk, two bytes of which are used by the hardware. Of the remaining six bytes, three are used for the coordinates of the destination, and the remainder for control information such as packet length, priority, etc. The remainder of a packet is the data payload, a maximum of 240 bytes (eight chunks minus the packet header). In the current implementation, each FIFO can hold 32 chunks. There is an interrupt available for each FIFO, controlled by a *watermark* level; receive FIFOs can be set to interrupt when there are more than *watermark* chunks in the FIFO, injection FIFOs to interrupt when there are fewer than *watermark* chunks in the FIFO. A receive interrupt will not be triggered, however, until all the chunks of a packet are in the FIFO.

This description of the Torus has concentrated on the features germane to the topic of this paper and has skipped many interesting details. For more information see the

excellent papers in the IBM Journal of Research and Development.[1,12]

Conceptually, this seems straightforward from the O/S driver point of view — the receive side handles an interrupt by sucking the packet out of the appropriate FIFO into a buffer and passing the buffer on, and the transmit side looks for a FIFO with enough room for the given packet and stuffs it in. The processor cycle budget, the speed of the network, the small packet size, and requirements such as *do not drop a packet* coupled with the hardware flow-controlled nature of the network create some interesting challenges for a general-purpose O/S using an interrupt-driven driver.

At the user level the current driver presents a single directory containing the files `torus`, `torusctl`, and `torusstatus`. The `torus` file is a raw interface: torus packets are read and written as-is, including the headers used by the hardware for routing. The other files are used for debugging.

The torus network cannot broadcast a packet to all nodes. It does, however, have a limited multicast enabled by the `deposit` bit in the torus packet header. This instructs the network to deposit a copy of the packet on each of the nodes it traverses on route to its destination; the route must be a straight line (in one dimension). Thus, a broadcast can be simulated with the assistance of the traversed nodes by, on receipt of a packet from the x or y direction with the deposit bit set, retransmitting the packet in their y or z dimensions respectively.

Packets injected into the network must have the header carefully checked for validity: invalid packets can cause an unrecoverable error rendering the entire network unusable until reset. Header check errors include destination coordinates outside the dimensions of the network, and hints as to the direction the packet should take leaving the node being inconsistent with the routing, dimensions or deposit information in the header. Because all our I/O is mediated by the kernel driver, we can do the checks, and safely share the device between applications and the system.

The FIFOs are 128 bits wide and can be loaded only by using the special BG/L floating-point unit, which has instructions for 128-bit load and store. A downside of this is the state of the floating-point unit, if in use by a process, must be saved and restored when used for FIFO stuffing by the kernel; single-process kernels do not have this restriction as they always know the entire state of the machine.

Class Routed Network Interface

The Class Route Network Interface, commonly called the 'Tree', is a special purpose network designed to support communication between groups of nodes, similar to the point-to-point and collective operations of MPI.[2] On BG/L, it also is the only messaging network that connects a group of CPU nodes to its I/O node: the I/O node is not on the Torus, and the CPU nodes are not on the Ethernet. All file I/O and other communications to the outside world must go over the Tree at some point.

The hardware network interface on each node has four ports. One port is dedicated to the local processors, and up to other three ports are wired to other nodes. Each port has a routing table, and each routing table entry provides different rules depending on whether the port is a source or a destination for a packet. The network is usually wired to form a trinary tree. The network carries packets with a 4-byte header interpreted by hardware and a 256-byte payload.

Class-based routing, as the name implies, differs from the normal destination-based routing used by a network such as Ethernet or the Torus. Instead of identifying a specific destination node, packet flow is determined by a 4-bit *class* field in the packet header. When a packet arrives at an interface over an external link, the network hardware uses the class as an index into the node's 16-entry route table. The table entry has two fields, one for incoming packets and one for outgoing packets. Each field tells on which outgoing links to forward the packet, including the local CPU. The routing

table can express many policies and topologies, within the constraints of the physical wiring, and the table entries will typically differ from node to node.

One useful convention is to reserve a pair of well-known class numbers to represent 'up' and 'down' links in a virtual tree. One node is chosen as the root (typically the CPU node with a link to the I/O node). At each node, the link to a neighbor closer to the root is designated the 'up' link, and the others are 'down' links to the nodes below, its children in the tree. The general pattern is that nodes send packets up to the top of the tree, and a response packet is sent back down to many nodes efficiently. We can use that for instance to multicast file contents.

The network supports two types of packets: point-to-point messaging, usually between pairs of nodes; and global computation and collective messaging using *combining* packets. Point-to-point packets carry a 24 bit value that can be matched against a per-node value to decide whether to accept the packet locally (forwarding is determined by the class in any case). Combining packets are more interesting. They specify a commutative operation — NOP, OR, AND, XOR, MAX or ADD — and a payload value up to 256 bytes long. When a combining packet is received, the hardware holds it until combining packets have arrived on all other links specified by the route table entry for the packets' class. The hardware then performs the operation, and sends the result packet on specified destination links. The routing tables are typically arranged so that values are sent from each node, combined under the operation as they move up the tree and, at the root, the resulting value is sent back down the tree to all nodes, unchanged. (The routing entry tells which direction should do the computation.) Thus, the Tree supports a global reduction capability, in hardware, without requiring each node to match up the packets and compute the result. Other than for global reductions, which represent a tree-based communications pattern, it is unusual to use the Tree between nodes; the Torus is a far better network for arbitrary communication.

The Tree has two almost independent interfaces, called virtual channels ('VCs'). The lowest-level device driver interface to the tree represents virtual channel 0 by the following files:

`vc0`	send and receive data (the driver adds the packet header)
`vc0status`	status, statistics (and register snapshot for development)
`vc0ctl`	set destination and class for packets
`vc0tag`	write to inject collective operations
`vc0p2p`	write to inject point-to-point packets

A similar set of files represents `vc1`. This representation will become more elaborate as we begin to multiplex protocols on a channel. This simple interface is just enough, however, to share the network between system and an application, with the system using it for Internet support, discussed next.

Internet protocols

Recall that CPU nodes are connected by the Torus and the Tree, that I/O nodes are connected to the CPU nodes by the Tree (not by the Torus), and that only the I/O nodes have a conventional network interface through Ethernet. Connections from the CPU nodes must go through the Tree and an I/O node to reach the Internet, for instance for file service (neither I/O nodes nor CPU nodes have any local permanent storage). Although the Tree must be used to get to an I/O node, the Torus is more efficient as a general-purpose messaging network, because of its address-based routing scheme. We therefore made the Internet protocols run on all three networks, which Plan 9 made easy to do, as follows. (In a later section we consider support for other protocols.)

The organization and implementation of networks in Plan 9 has changed somewhat since it was first described,[11] although the changes do not affect ordinary applications.

In particular, its implementation of the Internet Protocols has been significantly redesigned. It provides a model for our implementation of new HPC-assisting protocol stacks for the tree and the torus. A given kernel can have many independent IP stacks, each represented by a hierarchy of synthetic files, as shown in Figure 3.

```
/net/
          arp
          ...
          ipifc/
                    clone
                    stats
                    0/
                              ctl
                              data
                              err
                              listen
                              local
                              remote
                              snoop
                              status
                    1/
                              ctl
                              data
                              ...
                    ...
          iproute
          tcp/
                    clone
                    stats
                    0/
                              ctl
                              data
                              err
                              listen
                              local
                              remote
                              status
                    1/
                              ...
                    ...
          ...
```

Figure 3. Subset of `/net` on a node with several Internet network interfaces.

The IP stack multiplexes both interfaces and protocols, and this is reflected in the resulting name space, following the Plan 9 conventions for multiplexors. Thus IP protocols such as ICMP, TCP, UDP, and RUDP (Plan 9's reliable datagram protocol) have separate protocol directories that multiplex conversations on different addresses and port numbers. Each stack is controlled by reading and writing appropriate files in its name space. For instance, the current set of routes can be found by reading a table in plain text from `iproute`, and can be changed by writing a control message to it, also as text. Access to any file is controlled by the usual file permissions, implemented by the IP device driver.

The directory `ipifc` represents all the *interfaces* between the stack and the outside world, each corresponding to a particular set of IP addresses for the host within that stack. A new interface is added using the `clone` file in the usual way, then writing a control message that *binds* the interface to a particular *medium* that is the source and sink for packets on that interface.

The IP implementation is layered internally: one driver provides the name space and services for all protocols; below that, protocol drivers, represented by a type `Proto`, manage the state and messaging for each protocol; and at the lowest level media-specific code, represented by a type `Medium`, provide different ways to connect the IP stack to the outside world (ie, different MAC types). The scheme is highly modular. For

instance, the common IP level fragments packets using parameters provided by a given medium, but it leaves physical address management to that medium. Thus the details of ARP management and messaging, and Ethernet multicast, are found only in the Medium for Ethernet. The medium does not drive the hardware, but conveys the packets through a separate device driver or program, such as /net/ether0. Each component can present control and data files within the user accessible namespace, providing for varying degrees of access and control, for IP interface, protocol, and MAC levels.

Given the basic device drivers described above for the Tree and Torus, providing Internet access required writing two small Medium drivers, one for each network since they have different physical packet structure and addresses. Each medium driver receives outgoing IP packets from the stack, computes the appropriate Tree or Torus address (an easy function of the IP address), adds the network-specific header, and writes the result to the network's data file, named and opened during configuration of the interface. In the other direction, a kernel process in the medium driver continually reads from the network data file, strips the network-specific header, and with a function call pushes the packet up the IP stack. These medium drivers are similar in structure to the one for Ethernet, but much simpler, because they do not need to manage ARP for IP address resolution, nor do they need to handle multicast or both IPv6 and IPv4. Neither driver is more than 250 lines including comments and boiler-plate. The following extract from treemedium.c's bwrite implementation gives the flavor, where b is an outward-bound data Block:

```
th = (Th*)b->rp;     /* tree and IP header space in outgoing Block */

/* if we are going CPU to CPU, then we need class CPUClass. */
if((th->ipv4dst[1] != 0x80) && (th->ipv4src[1] != 0x80))
          class = CPUClass;
else
          class = IOClass;
if(th->ipv4dst[3] != 0xFF){
          /* calculate tree point-to-point address from IP address */
          p2pdst = xyztop2p(th->ipv4dst[1], th->ipv4dst[2], th->ipv4dst[3]-1);
          hdr = MKP2P(class, p2pdst);
}else
          hdr = MKTAG(class, 0, PIH_NONE);

th->hdr[0] = hdr>>24;
th->hdr[1] = hdr>>16;
th->hdr[2] = hdr>>8;
th->hdr[3] = hdr;

devtab[r->mchan->type]->bwrite(r->mchan, b, 0);
```

During system initialization, Plan 9 runs a shell script called /bin/cpurc that sets up the environment for the services to run on a given node, including network interfaces and daemons. On BG/L the script sets up new interfaces for the Tree and the Torus in the primary Internet stack. (We use the Tree as the example here; initializing the Torus IP interface is similar.) Cpurc invokes the following script to initialize the Tree:

```
#!/bin/rc
# Reading xyzip returns a sequence of 4 fields:
#         node's (x,y,z) coordinates, net mask,
#         gateway (x,y,z), I/O node p2p addr
xyzip=(`{cat /dev/xyzip})

# if it's an I/O node (coordinates are -1) skip this part
if(~ $#xyzip 4 && ! ~ $xyzip(1) '-1.-1.-1'){
          # create a new IP interface, and set i to its number
          i=`{cat /net/ipifc/clone}

          # bind the new interface to the file it will use for packet i/o
          echo bind tree /dev/vc0 > /net/ipifc/$i/ctl

          # Add the two tree routes
          echo add 11.$xyzip(1) $xyzip(2) 11.$xyzip(3) >> /net/ipifc/$i/ctl
          echo add 12.$xyzip(1) $xyzip(2) 12.$xyzip(3) >> /net/ipifc/$i/ctl
}
```

The script can use existing, simple commands such as `echo` or `cat` because parameters can be found in the name space as text files, and control messages are also written as text. For example, by convention, the node's peer-to-peer address on the Tree is derived from the node's (*x*, *y*, *z*) coordinates on the Torus, since those are unique. The script can find that data in `/dev/xyzip` because a certain device driver presents binary values taken or derived from the node's personality text form, in synthesized files bound to `/dev`. The script creates a new interface in `/net/ipifc` by following the usual conventions for `clone`, described above. The text command to bind the new interface to the existing device `/dev/vc0` (the data file for virtual channel 0 of the Tree, mentioned earlier), using the `tree` medium. The next two `echo` commands set the IP addresses for that interface, using node-specific values obtained earlier from `/dev/xyzip`. Special tools such as Unix's `ifconfig` and `route` are not required: control requests and network addresses are plain text, not awkward binary structures.

Coordinating initialization steps

In order for the Tree or Torus to be used, it must be *trained*: a known pattern is put on every link to allow each end to adjust the capture point of the incoming signal to compensate for the unknown phase relationship between the ASIC clock edge at either end. In order for training to work correctly, both ends of a link must at some point be training it together. At first, we thought that we could have each node train independently. We found that CPUs, after training, could converse with other CPUs, and the CPU nodes could successfully transmit to the pset's I/O node, but the I/O node could not send messages to the CPUs. It turned out that the original plan concealed a race. All the nodes are reset at once, and the CPU nodes (being identical) did tend to train their links together within a suitable interval, but the I/O node kernel was slightly different, because it configures an Ethernet, and the timing was off by just enough that the I/O node training was too late to catch the CPU's.

The kernel now uses the barrier network to ensure that all nodes start training at once. Indeed, we probably should be using a second barrier to ensure that all nodes have stopped training before any use is made of the network. The rules of parallel programming are not just for applications.

The plot so far

Everything described above has been implemented. Once the raw devices were working for the tree and torus, it was the work of perhaps an afternoon to write the `Medium` code to connect them to the IP stack. (We spent much more time dealing with the actual hardware.) It also saved time being able to configure the networks using the shell.

Given the resulting network connectivity, we can use all the Plan 9 services for distribution, including `cpu` service, `import` and `exportfs`. For instance, we can easily do

remote debugging, both at the normal process level, and at the core and JTAG level using the management name space described earlier. Our debugging sessions are controlled by x86 machines; the target processes are on PowerPCs.

Performance tests of the torus were disappointing. The BG/L processor is a 700 MHz PowerPC, and the CPU must push and pull every packet through the tree and torus FIFOs, but it can struggle to keep up. We took cycle-level measurements of the path from user-space to and from the wire. We now know that the kernel's `malloc` and spin lock implementations are too slow, partly because there is plenty of error-checking; and the queued block subsystem used by the networking code also must be reconsidered.

The packet payloads in both tree and torus networks are 256 bytes at most. Forcing user applications to fragment larger messages is inefficient. We have designed and are implementing a simple fragmentation scheme for both networks, allowing messages to be fragmented as they are placed on the network, and coping with the out-of-order delivery of the torus. No network is perfectly reliable, but given the high reliability of these networks, we can adapt ideas from various optimistic transfer protocols to improve performance.[13,14]

Related Work

The most obvious comparisons are with Linux and the BG/L Compute Node Kernel (CNK) on similar hardware. The CNK[1] supports only single-threaded applications, because HPC applications historically have not been concurrent programs. The CNK's set of system calls is an extended subset of the Linux system calls, to allow significant Linux applications to run unchanged. System calls that need full Linux kernel support, such as file or network I/O, are converted to remote procedure calls to the Linux system running on an I/O node. Applications access the torus directly. The Blue Gene MPI implementation can make use of the tree to do collective operations, when the context and parameters are suitable. The CNK is generally the most efficient kernel on the Blue Gene. We do not expect to equal its performance, but we must be within (say) 10% of it, or the cost to applications might be too high. In exchange for some loss of raw performance on a given node, we shall provide infrastructure that improves overall system performance, and makes it easier to manage a program with tens of thousands of processes.

Linux provides a single Internet stack, accessed through the BSD socket interface.[15] Superficially, Linux has a similar split to Plan 9 between the IP stack and devices that carry IP packets, through its `net_device`. That interface is more complex, however, and indeed the introductory comment reads:

```
/*
 *      The DEVICE structure.
 *      Actually, this whole structure is a big mistake. It mixes I/O
 *      data with strictly "high-level" data, and it has to know about
 *      almost every data structure used in the INET module.
 *
 *      FIXME: cleanup struct net_device such that network protocol info
 *      moves out.
 */
```

A `net_device` has many operations, and refers to VLANs, bridges, and particular protocols. They are hard enough to write correctly that unusual networks often make themselves look like an Ethernet, inventing a meaning for ARP, just so the existing Ethernet `net_device` can be used unchanged to link the new network to the Internet stack. This hints at another significant difference: a Plan 9 `Medium` driver is not, in fact, a 'net device' itself but just an interface within the IP implementation, from IP to another self-contained device in the name space. The medium opens the target device in much the same way as any other client, and reads and writes the resulting file descriptor. This provides complete insulation between the medium and device, and better modularity. In

our case, we also wish applications to access the network device independently of IP, for instance to access the global reduction operations of the Tree.

Plan 9's handling of addresses also reduced our work. The BSD socket system calls declare a `struct sockaddr` as the network address for `bind`, `connect` and `accept`. Because the format of the structure changes with the network type, that declaration is misleading. There is actually a family of such structures: `sockaddr_in` for Internet addresses, `sockaddr_un` for UNIX domain sockets, `sockaddr_x25` for X.25 addresses, and so on. These differ radically in size, format and content. The particular format is determined by an initial 16-bit value, which must be globally unique, making it hard to add a new one. In practice applications tend to be limited to the types and formats declared in their source code. Ideally, a plain `sockaddr` would be big enough for all types (and how big is that?), but on Linux it is not.

By contrast, Plan 9 represents all addresses as text, with a standard syntax specifying *network* (protocol), *host* and *service*, using either logical or physical addresses. Applications rarely interpret addresses directly. Instead, a general *connection service* translates addresses into recipes for obtaining network connections, exploiting Plan 9's conventions for network name spaces. For instance, the *network* component names the multiplexor directory for the network or protocol. The *host* and *service* are interpreted by the specific network's control file. The combination of text format and name translation ensures that applications are indeed network independent; no code need be changed or even recompiled to add new networks.

Distributed computations often use *one-sided* communication: instead of full-duplex conversations, messages are sporadically sent in one direction only. For example, a controlling process might send messages to other CPU nodes to provide parameters for the next phase of a computation; much later, those nodes might send back the results. The CNK implements a special protocol for such cases.[16] When sending large messages as many packets, it includes full context data in initial packets to reduce latency until a context has been established with the receiver, allowing subsequent packets to quote that to increase bandwidth.

Many HPC applications today use the messaging framework of MPI.[2] It provides point-to-point and collective messaging between groups of processes. It offers an explicit choice of messaging strategies, and an elaborate infrastructure, including the dynamic definition of types for marshalling data between heterogeneous systems. Notably, it does not define any particular protocol. We shall provide a lightweight MPI implementation for Plan 9, to support existing applications unchanged. Even so, given the support of a modern distributed operating system, we wish to revisit some of the design choices behind MPI, to provide a leaner, simpler interface for distributed computation. For example, Ron Minnich quickly developed a tiny messaging library for Plan 9 on Blue Gene, with just four primitives: *send*, *receive*, *reduce* and *barrier*, initially implemented for the Torus. (They access the underlying network by reading and writing a file descriptor.) He has modified several important HPC benchmark programs to use it.

Further Work

We have a useful initial port of Plan 9 to the Blue Gene. Now the real work begins. The work items in the larger project include: design and implementation of services that aggregate many nodes into a comprehensible unit; improved fault-tolerance and reliability (as the density of cores increases, and computations last longer, fault rates will increase); and better system management. All of those rely on good handling of the various interconnects.

When many nodes start an application, or move from phase to phase within it, they can issue many requests for the same files, in our case using the 9P file service protocol. We plan to use the multicast ability of the Tree to answer many file requests at once.

For that and other applications, we are changing the Tree and Torus drivers described above to multiplex a handful of conversations, typically representing different protocols, for instance IP and one-sided messaging on the Torus, or IP and multicast 9P on the Tree. The software header we defined for fragmentation also allows multiplexing of several higher level protocols on the network. Some of those protocols might have their own conversations (similar to having several Ethernet protocols, one of which is IP). We are especially interested in lightweight protocols that can take advantage of the special properties of networks like the tree or torus, more effectively than IP. O'Malley and Peterson showed that a complex protocol graph of simple protocols could outperform a simple graph of complex protocols,[14] and we might borrow some of their ideas here, perhaps composing specialized lightweight protocols to suit particular applications.

We noted above that an application needs the same barrier network, in the same mode, on all nodes. On the Tree, the use of the collective operations on a given virtual channel also must be coordinated, so that a given application owns the channel system-wide for the duration of a sequence of operations, or the network will (in effect) deadlock or malfunction. We need higher-level agreement services to allow that, and some controls in the driver to enforce it.

We continue to work on performance improvements in kernel and network subsystems, which should benefit Plan 9 generally, not just in the HPC realm.

Acknowledgements

This work has been supported by the Department of Energy Office of Science Operating and Runtime Systems for Extreme Scale Scientific Computation project. The port of Plan 9 to the Blue Gene hardware would not have been possible without the support of IBM's Blue Gene research team as well as contributions and support from Volker Strumpen, Amos Waterland, Volkmar Uhlig, and Jonathan Appavoo.

References

1. A Gara, M A Blumrich, D Chen, G L-T Chiu, P Coteus, M E Giampapa, R A Haring, P Heidelberger, D Hoenicke, G V Kopcsay, T A Liebsch, M Ohmacht, B D Steinmacher-Burow, T Takken, P Vranas, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development* **49**(2-3), pp. 195-212 (2005).

2. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard,* University of Tennessee, Knoxville, Tennessee (1995).

3. Ronald G Minnich, Matthew J Sottile, Sung-Eun Choi, Erik Hendriks, Jim McKie, "Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels," *ACM SIGOPS Operating Systems Review* **40**(2), pp. 22-28 (April 2006).

4. Ron Minnich, Jim McKie, Charles Forsyth, Latchesar Ionkov, Andrey Mirtchovski, Eric Van Hensbergen, Volker Strumper, *Rightweight Kernels*, http://www.cs.unm.edu/~fastos/07meeting/usenix-june2007.pdf.

5. Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom, "Plan 9 from Bell Labs," *Computing Systems* **8**(3), pp. 221-254 (Summer 1995).

6. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, "The Use of Name Spaces in Plan 9," *Proceedings of the 5th ACM SIGOPS European Workshop*, Mont Saint-Michel (1992).

7. C H Forsyth, "The Ubiquitous file Server in Plan 9," *Proceedings of the Libre Software Meeting*, Dijon, France (2005).

8. Phil Winterbottom, "Acid: A Debugger Built From A Language," *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, California, pp. 211-222 (1994).

9. *Plan 9 Programmers Manual, Fourth Edition (Manual Pages),* Bell Labs Lucent Technologies (2002).

10. Matthew J. Sottile, Ronald G. Minnich, ''Supermon: A High-Speed Cluster Monitoring System,'' *IEEE International Conference on Cluster Computing (CLUSTER'02)*, p. 39 (2002).

11. Dave Presotto, Phil Winterbottom, ''The Organization of Networks in Plan 9,'' *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, pp. 271–280 (1993).

12. N R Adiga, M A Blumrich, D Chen, P Coteus, A Gara, M E Giampapa, P Heidelberger, S Singh, B D Steinmacher-Burow, T Takken, M Tsao, P Vranas, ''Blue Gene/L torus interconnection network,'' *IBM Journal of Research and Development* **49**(2–3), pp. 265–276 (2005).

13. John B Carter, Willy Zwaenepoel, ''Optimistic Implementation of Bulk Data Transfer Protocols,'' *Proc. 1989 ACM SIGMETRICS and PERFORMANCE '89: International Conference on Measurement and Modeling of Computer Systems* (1989).

14. Sean W O'Malley, Larry L Peterson, *A Dynamic Network Architecture,* University of Arizona.

15. Marshall Kirk McKusick, George V Neville-Neil, *The Design and Implementation of the FreeBSD Operating System,* Addison-Wesley (2004).

16. M Blocksome, A Sidelnik, C Archer, B Smith, S Krishnamoorthy, T Inglett, G Almási, P McCarthy, J Castaños, V Tipparaju, M Mundy, D Lieber, J Nieplocha, J Ratterman, J Moreira, ''Design and Implementation of a One-Sided Communication Interface for the IBM eServer Blue Gene Supercomputer,'' *SC2006*, Tampa, Florida (November 2006).